Twisted Python – Proxy Server Herd
UCLA Computer Science 131: Programming Languages

Anthony Mirand-Vidaurre
University of California, Los Angeles

Abstract

The event-driven networking engine Twisted was used to prototype an application server herd implementation. This prototype involved the creation of a five server network, each in communication with a pre-defined subset of the network. The servers process client location updates and client location information requests. The servers also send client location reports to other directly connected servers by means of a flooding algorithm. Each server logs all connections, transmissions, and error messages in respective log files. This report describes the design and implementation of the prototype, possible considerations or alternatives from the design process, and topics covering programming language-related concerns.

1 Introduction

"Twisted is an event-driven networking engine written in Python and licensed under the open source MIT license. Twisted runs on Python 2 and an ever growing subset also works with Python 3." [1]

Given the task of implementing an application server herd that supports sending updates quickly to all servers to maintain a consistent cache initially seemed like a tall task, but being required to prototype such an application using the Twisted engine provided some direction. Additionally, choosing an engine that was written in Python helped simplify some concerns writing a complicated networked application. This report will continue to highlight the decision decisions and implementation details for this server herd prototype. This report will then explore a feasibility analysis and subsequent programming language comparisons with Node.js.

2 Motivation

There are many stacks upon which web based architectures are built upon, such as LAMP (Linux, Apache, MySQL, PHP) or MEAN (MongoDB, Express, Angular, Node.js). However, for some kinds of data operations that require quick and in-sync updates, such an architecture with central database communication might not be worth the lack of efficiency. Futhermore, to design a service that operates under the assumption of rapidly-evolving data/updates, various web protocol support, and mobile clients, a stack with central database communication seems more out of place. To address the issue with central database communication, the Wikimedia architecture could be a reasonable choice with its LAMP stack and multiple redundant web servers behind a load-balancing virtual router for reliability and performance [2]; but the application server from the Wikimedia architecture imposes a bottleneck due to the server overhead of adding new servers, which is a concern for this type of application.

The potential of using an application server herd, for which the servers act as intermediary proxies between each other and can push herd-wide updates to keep track of client updates, seems like a useful alternative. This is where the Twisted event-driven engine might be useful.

3 Design

This section of the report focuses on both the specified behaviors (protocols) and unspecified behaviors (multiple roles of servers and inter-server communication) that went into designing a proxy server herd application.

3.1 Protocol behaviors

Each server in the server herd should accept TCP connections with neighboring servers and clients. Once a client connects to a server, clients should be able to send messages to the connected server. Clients can update a server with their current location, in which the connected server should acknowledge the client by echoing back the processed client's location. Clients can also query location information regarding other client's locations, in which the connected server is to acknowledge the information request and query the Google Maps Platform Places API [3] on behalf of the client, all of which is sent back to the client. Internally and hidden to clients, servers should be able to propagate client location updates to other connected servers that they are allowed to communicate with, defined by a configured network topology. [4]

We formalize the three aforementioned protocols (location updates, location information requests, and location

propagating or flooding) as IAMAT, WHATSAT, and AT messages respectively. [4]

Lastly, each server logs its input and output to a log file, alongside server startup, teardown, and error messages.

3.2 Multiple roles of servers

In the network, each node in the herd serves as a server and a client. The node acts as a server when location update and location information request queries are made; contrastingly, the node acts as a client when propagating location statuses throughout the network. Since each node therefore has two responsibilities, an argument could be made to split each node into disjoint server and client components. To achieve this we could have a node listen on different server and client ports. However, this configuration adds unnecessary complexity to this prototype so we opted for a node listening a single port.

We can differentiate server and client roles instead by parsing the message we receive: client-server protocols consists of IAMAT (updating client location) and WHATSAT (querying information near other clients' locations), server-client communication consists of AT (responding to location updates or location information requests), and server-server communication consists of AT (propagating locations throughout the network.)

3.3 Inter-server communication

One main concern of client-server interactions was response accuracy. When a server receives a request from a client, whether it be a location update or a location query, the server must reply with the most up-to-date data. Thus, the client's locations must be synchronized across the network in order to provide reliable information.

There are two approaches that were considered to achieve this synchronization:

1. Only store client location information at the server which the client is connected and communicating with and propagate data on demand. This approach will introduce a larger response time delay if the server does not have a specific client's information on hand and needs to send a signal to other servers to fetch the data. However, this has an advantage in that the amount of data stored at each server will be smaller and distributed, since servers are only responsible for maintaining data for its own client connections. In a network that expects a lot of queries and updates, this approach does not seem to be the most efficient.

2. Store all client location information at every server in the herd, and immediately flood the herd when a server receives a client location update. The approach trades a space disadvantage for a speed advantage since all servers in the herd should have all clients' information (if the network topology does not define any disjoint sets). However, the space disadvantage that this approach imposes is not very significant since this type of application is concerned with managing rapidly-evolving text data in an in-memory cache. In a network that expects a lot of queries and updates, this approach seems to be the most efficient.

Based on this prototype's use case, we opted to use the second approach. When a client makes a location update, the update in flooded across the network so the other servers are aware of the changes.

3.3.1 Flooding algorithm

Since we model server-server communication using AT messages, we can reference the format of AT messages in 4.4 (copied below for quick reference):

```
AT <server_id> <time_difference>
<client_id> <coordinates> <client_time>
{<sender_server_id>}
```

When using AT messages for server-client communication, we omit the optional <sender_server_id> argument, but this is pivotal for server-server communication. We include the sending server's id so that the receiver of this messages knows not to flood the message back to the server they received it from. However, this alone cannot eliminate all infinite location update loops in a network topology where cycles with more than nodes exists. Thus, we introduce the notion of duplicate or out-dated AT messages. Used in conjunction with the sending server name exclusion, if a server receives an AT message that is not more recent than an AT message for the client already in its cache (denoted by the timestamp), the server does not propagate the location update to other servers it is connected to.

It is also worth mentioning that the two approaches have varying impacts on inter-server communication overhead; for example, in order to recover data from a server failure from a program crash, loss of internet connection, or temporary loss of power, the second approach requires the server to proactively synchronize with available servers, while the first approach does not. Given the time frame of this prototype however, it does not implement a proactively-synchronizing data recovery method.

We could attempt to hack together a naive data recovery method by trapping upon server teardown and serializing the data to disk while the server is shutting down. However, since this prototype focuses on rapidly-evolving data/location updates, we opted to not serialize old data as it will likely be overwritten when the server reconnects and is sent new updates.

This prototype does have a simple connection recovery method where connected servers attempt to reconnect and re-propagate newly-received data to lost servers. Each server keeps a list of currently-connected servers and used the stored server protocol to propagate client location updates. If a connection is broken or interrupted, instead of using the connected server list, the server attempts to replace the stored server protocol with a new TCP connection each time a client location needs to be propagated.

4 Implementation

With the design details considered, this section aims to cover the mechanics of the server herd. All concrete code of the mentioned objects and protocols can be found in at the end of the report or online at `https://www.github.com/anthonymirand/archive/proxy-server-herd/`.

4.1 Architecture

The Twisted engine provides interfaces, protocols, and factories that help make it easy to implement custom network applications.

To begin, all Twisted applications of this nature must have a continuously listening loop in the form of a `twisted.internet.reactor`. The reactor handles connections on a specific address and port using a particular protocol.

A protocol is used to define how a server parses network messages, and a factory is used to create a protocol and maintains the persistent configuration. Specifically, we can leverage the `twisted.protocols.basic.LineReceiver` and `twisted.internet.protocol.ServerFactory` to classes create the server protocol and server factory respectively. The protocol will inherit from `twisted.protocols.basic.LineReceiver` in order to setup a line-based, or text data separated by line breaks, form of message communication. [5]

Because servers must also behave like clients (as discussed in 3.2), we chose to also implement a lightweight client class; more specifically, using the same `LineReceiver` protocol as the server above but instead using the `twisted.internet.protocol.ClientFactory` class. This was done to create a simplified interface for a server to communicate with a connected server/"client"[1].

Lastly, each running server will maintain a cache of location updates it has received from clients and connected servers. For simplicity, we chose to use Python's `dict()` to keep a map of client id to the most recent server confirmation (`AT` message), client position, and client time.[2]

4.2 Handling `IAMAT` messages

`IAMAT` messages can be sent from a client to a server to indicate a client's location at a particular time. The established protocol is that all `IAMAT` messages should obey the following format:

```
IAMAT <client_id> <coordinates> <client_time>
```

where the first field is the name of the command that signifies the client is telling the server where it is, the second is the requesting client's id, the third is the current client's latitude and longitude, and the fourth is the client's idea of when it sent the message.

When processing this message, the string must contain four fields, the coordinates must be in decimal degrees using ISO 6709 notation [6], and the client time must be a valid POSIX timestamp. We validate this input before performing any work by using the `utils.validate(validationHandler)` decorator with handler `utils.ValidateIAMAT(data)` that throws a `utils.ValidationError` exception if any part of the message is malformed.

Upon receiving a well-formed message, we echo an `AT` response to notify a successful transmission. If the client id exists in the server's cache, we check to see if the timestamp for the client id is more recent than the cached version. If not, we simply discard this message internally and return.[3]

The server then propagates this location information (the returned `AT` message, client location, and client time) to other connected servers in the herd. This process will append the current server's id to the original `AT` before initiating the location updating flooding, as mentioned in 3.3.1; the propagation will conclude when the servers receive duplicate client location updates and are then configured to no longer propagate the message.

4.3 Handling `WHATSAT` messages

`WHATSAT` messages can be sent from a client to a server to request information about places near other clients'

[1] At this point in the report there are two definitions of clients: users who make location updates/requests and other servers that a "target" server is connected to. This confusion could be amplified with the notion of "clients" being maintained in the appendix code, which should rather carry the semantics of a connected server. For the rest of the report, all references to "clients" will be in terms of making requests to a connected server, where connected servers will be referred to as "servers in the herd" or "connected servers".

[2] Use of an in-memory data structure as a location cache was mentioned in 3.2 in the naive data recovery method paragraph.

[3] An argument could be made that we should inform the user of the outdated or duplicate information, but since we also use this logic to flag a flood termination, we chose not to introduce an extra layer of complexity.

locations. The established protocol is that all WHATSAT messages should obey the following format:

```
WHATSAT <client_id> <radius> <bound>
```

where the first field is the name of the command that signifies the client is requesting location information, the second is another client's id, the third is a radius in kilometers from the provided client, and the third is an upper bound on the amount of location information to receive from the request.

When processing this message, the string must contain three arguments, the radius must be an integer in the range 0–50, and the bound must be an integer in the range 0–20. We validate this input before performing any work by using the `utils.validate(validationHandler)` decorator with handler `utils.ValidateWHATSAT(data)` that throws a `utils.ValidationError` exception if any part of the message is malformed.

Upon receiving a well-formed message, we check to see if the server has any client information for the requested client in its cache. If not, we inform the client that the server cannot complete the request since it has no knowledge of the client (namely its location), and returns.

The server then instantiates a callback in the form of a GET request to the Google Maps Platform Places API with the client's location and included search radius. If this request is successful, we limit the number of entries in the API response as per the provided bound, and echo an AT response to notify a successful transmission alongside the API response JSON information; if there was an error in receiving Google Places data, we inform the client of the API request error and return.

4.4 Handling AT messages

AT messages can be sent from a server to a client to acknowledge a successful message transmission, or from a server to a server to propagate client location data. The established protocol is that all AT messages should obey the following format:

```
AT <server_id> <time_difference> <client_id>
<coordinates> <client_time> {<sender_server_id>
| <data>}
```

where the first field is the name of the response or the command that signifies a location update propagation, the second field is the id of the server that received the message from a client, the third field is the difference between the server's idea of when it received the message and the client's timestamp, the fourth field is the requesting client's id, the fifth is the current client's latitude and longitude, and the sixth is the client's idea of when it sent the message. There is an optional seventh argument that either contains the id of the server that sent the message (in the case of server-server communication via AT) or the requested location information (in the case of server-client communication via WHATSAT) in the form of a JSON response; there is no need for this optional argument when a server is simply acknowledging a successful message transmission (in the case of server-client communication via IAMAT).

When processing this message, the string must contain seven arguments (since this message can only be received in the event of server-server communication), the time difference must be a valid POSIX time difference, the coordinates must be in decimal degrees using ISO 6709 notation [6], and the client time must be a valid POSIX timestamp. We validate this input before performing any work by using the `utils.validate(validationHandler)` decorator with handler `utils.ValidateAT(data)` that throws a `utils.ValidationError` exception if any part of the message is malformed.

Since servers can only receive AT messages in server-server communication, we will only explain AT message's propagation logic.

Similar to handling IAMAT messages, if the client id exists in the server's cache, we check to see if the timestamp for the client id is more recent than the cached version. If not, we simply discard this message internally and return; this serves the dual-purpose of stopping a propagation cycle.

The server then propagates the received AT message to other connected servers in the herd. Contrastingly to the IAMAT propagation, this process does not append the current server's id to the original AT message. This is because upon receiving an IAMAT message, the IAMAT-receiving server added their server id and initiated a flood; all other servers will then receive an AT message with the original server's id and we do not want to invalidate a form of loop detection.

5 Considerations

After describing the design objectives, design decisions, and implementation details, the following section illustrates some of the implementation issues and concerns.

5.1 Circular references

In a programming languages course, one topic of discussion is how object reference cycles are handled by the garbage collector; for example, how it might cause problems in a pure reference counting garbage collector. Even if imposing an issue with garbage collection, sometimes reference cycles are warranted for ease of design. We utilized circular references in the inter-server communication and aim to justify this design decision.

In this prototype, an object of type `twisted.internet.protocol.Factory` keeps a reference to an object of type `twisted.internet.protocol.Protocol` and vice-versa. This is done for convenience: in effect, an object is able to modify itself via a factory method depending on an input it receives via a protocol method, and the factory object can check with its protocol method if it needs to undergo such a modification.

As a way to manage the aforementioned connection recovery system, the node keep a record and maps its connection with other nodes in the network and the other nodes keep a record of the same connection; this map contains `client.ProxyClientFactory` objects that we defined in this prototype. All `client.ProxyClientFactory` objects in this map contain a reference to their respective `server.ProxyServerProtocol` objects. If a connection is either created or lost, the client object will detect it and have to update the server's connection map. Additionally, a server instance can call the `client.sendAT(at_message)` function on its stored client object to facilitate location update propagation, or can attempt to construct a new client object if it needs to create a new connection. This implementation manages to support the connection recovery system and only reconnects when a failure is detected, instead of opening and closing connections to send every `AT` message.

Python does have a reference counting based garbage collector that has the ability to detect reference cycles [7]. Therefore, it is unnecessary to implement a system to explicitly nullify a node in a cycle to prevent memory leaks or inaccessible objects for this server herd prototype. We are justified in creating circular references with the benefit of increased convienience flexibility.

5.2 Race conditions

Since Twisted is asynchronous and implements network communications, determining if the framework is data race free cannot be concluded immediately. We have to look into the Twisted source code to get a deeper insight.

5.2.1 Are Twisted callback functions atomic?

One immediate concern is determining if the execution of a callback function could be interrupted to execute another callback function, thus potentially introducing data races.

Examination shows that Twisted's callback functions are asynchronous due to a single-threaded implementation; the callbacks are called from the event loop (specifically `twisted.internet.reactor.run()`) which is a blocking function and runs only on one thread [8]. If another event occurs, instead of interrupting the current event's execution, it will be queued for execution. Therefore, due to the way Twisted's event mechanism is setup, Twisted callbacks are executed sequentially and do not allow for data races to occur.

5.2.2 What happens if two events occur simultaneously?

Consider the following scenario:

- client `A` queries server `B` for the location of client `C`
- at the same time, server `B` receives a client updates of client `C` from client `D`

The issue lies in determining if client `A` gets the old value of client `C`'s location or the new value sent from client `D`. Unfortunately, the results are undefined because a network is involved and it ultimately depends on the processing order of the messages. It is difficult to replicate such a scenario in this implementation, but believe it is fair that it is possible since this is an arguable unavoidable problem in distributed systems. Therefore, we conclude that this scenario could happen and is unavoidable.

With the previous two questions, we can conclude that the implementation is data race free since the main event loop executes callbacks sequentially, and there is no reliable control over the order in which the network handles simultaneous events.

6 Discussion

The following section describes the high-level considerations made when deciding to make a server herd using Twisted in Python, not directly relating to this prototype's specific design or implementation. This discussion is fitting for the scope of a programming language course.

6.1 Feasibility of prototyping with Twisted

Provided that a server herd was successfully prototype, a fair conclusion would be that is it feasible to write such an application with Twisted. Nothing that the author did not have previous experience writing networked programs in Python, it was not a significant challenge to prototype a server herd.

In the limited experience that the author has in creating networked applications at the time of this report (namely a small embedded systems project in an Operating Systems course), he does recall some of the intricacies of networking in C. In C's `socket` API, the API wraps system call functions such as `accept(2)`, `bind(2)`, `listen(2)`, `read(2)`, and `write(2)` [9]. As a relative beginner to

writing networked applications, the Twisted framework does provided a more abstracted and streamlined way to create networked applications. For example, Twisted provides a `connectionLost` callback that has less programmer overhead than checking `read(2)` return values to then debug a lost connection in C.

6.2 Feasibility of prototyping with Node.js

Node.js is a JavaScript runtime that can be run in an operating system from a console, as compared to JavaScript that is run inside a browser. Node.js uses an event-driven, non-blocking I/O model; similar to Twisted, Node.js presents an event loop but instead of as a library, as a built construct. [10]

If we were to implement a proxy server herd in Node.js, the architecture of the program is one thing that would need to change due to Node.js's language design.

Node.js is a prototype-based programming language[4], meaning that objects are cloned and reused to build applications. Additionally, there are no concepts of classes in JavaScript supporting Node.js, although ECMAScript 2015 introduced classes as syntactic sugar for the underlying prototype reuse and inheritance [11]. Instead of inheriting from classes like `twisted.internet.protocol.Factory` and `twisted.internet.protocol.Protocol`, we would have to define prototypes of a factory and protocol, and then use inheritance to create objects from the prototypes.

Given the robust networking capabilities and language constructs of Node.js, it is reasonable to assume that creating a server herd-like application in Node.js is feasible.

Additionally, functions are treated as function objects in JavaScript, thus given them more expressive power on the scale of objects. One notable difference compared to Twisted is that in place of Twisted's `twisted.web.client.GetPage(url)` we could declare a function object that could wrap the entire callback, processing, and error handling functionality.

6.3 Twisted/Python vs Node.js

In the following subsections, Python will be used in lieu of Twisted for the same of comparing two programming languages and assume that Python is running on top of the CPython interpreter [12].

6.3.1 Type checking

Python and Node.js both use dynamic type checking, which checks the types of objects during run time compared to compile time, but Python is a strong typed language whereas Node.js (JavaScript) is a weakly types language [13]. In dynamically checked languages, programs are generally simpler to write quickly and have less compile time overhead. On the other hand, since dynamically checking languages employ duck typing to resolve object types, it introduces a programming comprehension overhead since programmers need to recognize types without explicitly being told the types of objects in the program.

Comparing Python and Node.js in this regard is a fairly moot point due to the same system of type checking in both languages.

6.3.2 Memory management

In a similar vein of similar implementations, both Python and Node.js employ the use of garbage collectors as their choice of automatic memory management.

Python employs a mixture of reference counting and non-moving mark-and-sweep garbage collection; as previously mentioned Python's garbage collector can clean up cyclic data structures which is used in this implementation [7, 14, **?**], Once notable attribute about the Python Memory Manager is that there is no concept of a nursery, which does simplify the role of memory management [15].

Node.js is built on Google's V8 JavaScript engine [16]. This engine employs the use of a stop-the-world garbage collector, which halts all program execution during garbage collection, and makes use of scavenge and mark-and-sweep techniques on young and old generational objects respectively [**?**]. However, since the engine is written in C++, memory leaks are a valid and real concern with writing code in Node.js.

Since Python and Node.js do not allow direct memory access, they made the trade-off of remove the arguable flexibility of memory access in languages like C or C++ for ease of object creation.

6.3.3 Multi-threading

Although Python does support creating multithreaded applications, the use of Twisted in this project limits execution to one thread. On the other hand, Node.js does support multithreaded application (more specifically, the virtual machine can handle some operations in parallel [13]) and would not impose any restrictions to its use. With this in mind, Node.js could potentially outperform Python (using Twisted) for use in an application server herd. Additionally, since Node.js does already outperform Python in common code benchmark testing, it would not be surprising to see if the trends carried over to multithreaded benchmarks [17].

[4] Not to be confused with creating application prototypes.

Another hindrance of Python is that Python's memory management does not work well with multithreading. There is a need to introduce a Global Interpreter Lock that limits the true concurrency that any multithreading would bring to an application [18].

Since multithreading could be useful with asynchronous I/O and given the limitations of Python with Twisted in mind, Node.js has a slight upper hand for multithreaded server herd implementations.

Conclusion

The Twisted framework was used to prototype a proxy server herd application. The report covers the design choices and implementation details of the application, namely inter-server communication, circular references, and race conditions. The feasibility of using Twisted, proposed feasibility of using Node.js, and a comparison to Node.js follow, which conclude that both are acceptable approaches to creating server herd prototypes.

References

[1] Twisted Website. https://twistedmatrix.com/trac/.

[2] Wikimedia Architecture. https://upload.wikimedia.org/wikipedia/labs/8/81/Bergsma_-_Wikimedia_architecture_-_2007.pdf.

[3] Google Maps Platform Places API. https://developers.google.com/places/web-service/intro.

[4] Fall 2016 CS 131 Project Description. http://web.cs.ucla.edu/classes/fall16/cs131/hw/pr.html.

[5] Twisted Matrix Labs: Writing Servers. https://twistedmatrix.com/documents/current/core/howto/servers.html.

[6] Standard representation of geographic point location by coordinates. https://www.iso.org/standard/39242.html.

[7] Supporting Cyclic Garbage Collection. https://docs.python.org/2/c-api/gcsupport.html.

[8] Using Threads in Twisted: How Twisted Uses Threads Itself. https://twistedmatrix.com/documents/current/core/howto/threading.html#how-twisted-uses-threads-itself.

[9] Linux Programmer's Manual: socket(2). http://man7.org/linux/man-pages/man2/socket.2.html.

[10] About Node.js. https://nodejs.org/en/about/.

[11] JavaScript: Defining classes. https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Classes.

[12] GitHub: The Python programming language. https://github.com/python/cpython.

[13] Paul Eggert. UCLA 131 Programming Languages. Lecture.

[14] Python Garbage Collector interface. https://docs.python.org/3/library/gc.html.

[15] Python Memory Management. https://docs.python.org/2/c-api/memory.html.

[16] V8 JavaScript Engine. https://v8.dev/.

[17] Python vs Node.js Performance. https://benchmarksgame.alioth.debian.org/u64q/compare.php?lang=node&lang2=python3.

[18] Python Global Interpreter Lock. https://wiki.python.org/moin/GlobalInterpreterLock.

main.py

```
#!/usr/bin/env python

# CS 131 - Twisted Project
# Entry point for running servers (and clients)

import conf
from server import ProxyServerFactory

import sys, requests

from twisted.internet import reactor
from twisted.internet.error import CannotListenError


def usage(error=None, name_error=False):
  print "Usage: python main.py [server_name]"
  if error:
    print "ERROR: {}".format(error)
  if name_error:
    print "Valid server names are: {}".format(conf.SERVER_CONFIG.keys())
  exit()


def main():
  if len(sys.argv) != 2:
    usage()

  server_name = sys.argv[1]
  if server_name not in conf.SERVER_CONFIG:
    usage(
        error="{} is not a valid server name".format(server_name),
        name_error=True)

  try:
    factory = ProxyServerFactory(server_name,
                                 conf.SERVER_CONFIG[server_name]["port"])
    reactor.listenTCP(conf.SERVER_CONFIG[server_name]["port"], factory)
    reactor.run()
  except CannotListenError:
    usage(error="Port {} already in use".format(conf.SERVER_CONFIG[server_name]
                                                ["port"]))


if __name__ == '__main__':
  main()
```

conf.py

```
#!/usr/bin/env python

# Configuration file for the Twisted Places proxy herd

GOOGLE_API_KEY = "SECRET_KEY"

GOOGLE_PLACE_API_URL = "https://maps.googleapis.com/maps/api/place/" \
    "nearbysearch/json?location={}&radius={}&sensor=false&key=" + GOOGLE_API_KEY

SERVER_CONFIG = {
  "Alford"  : { "ip" : "localhost", "port" : 11500 },
  "Ball"    : { "ip" : "localhost", "port" : 11501 },
  "Hamilton" : { "ip" : "localhost", "port" : 11502 },
  "Holiday" : { "ip" : "localhost", "port" : 11503 },
  "Welsh"   : { "ip" : "localhost", "port" : 11504 }
}

SERVER_NEIGHBORS = {
  "Alford"  : [ "Hamilton", "Welsh" ],
  "Ball"    : [ "Holiday", "Welsh" ],
  "Hamilton" : [ "Holiday", "Alford" ],
  "Holiday" : [ "Ball", "Hamilton" ],
  "Welsh"   : [ "Alford", "Ball" ]
}
```

server.py

```
#!/usr/bin/env python

# CS 131 - Twisted Project
# Server Implementation

import conf, utils
from client import ProxyClientFactory

import sys, re, os, time, json, logging

from twisted.internet import reactor, protocol
from twisted.protocols.basic import LineReceiver
from twisted.web.client import getPage


class ProxyServerProtocol(LineReceiver):

  def __init__(self, factory):
    self.factory = factory
    self.at_response = "AT {}".format(factory.server_name) + " {:+f} {} {} {}"

  def connectionMade(self):
    self.factory.connections += 1
    self.infoHandler("CONNECTION ESTABLISHED; Total: {}".format(
        self.factory.connections))

  def connectionLost(self, reason):
    self.factory.connections -= 1
    self.infoHandler("CONNECTION LOST; Total: {}".format(
        self.factory.connections))
```

```
31
32    def commandFailed(self, line):
33      logging.error("INVALID COMMAND: {}".format(line))
34      self.transport.write("? {}\n\n".format(line))
35
36    def help(self, quit=False):
37      help_message = "\n".join((
38          "Available commands:",
39          "> IAMAT [client_id] [client_position] [client_time]",
40          "> WHATSAT [client_id] [radius] [bound]",
41          "> AT [server_id] [time_diff] [client_id] [client_position] [client_time]",
42          "> help", "> quit\n\n"))
43      quit_message = "\n".join(
44          ("To disconnect from the current session, use the Telnet escape",
45           "character '^]' (Control-]), and then type 'quit' into the Telnet",
46           "interpreter to safely close the connection.\n\n"))
47      self.transport.write(help_message if not quit else quit_message)
48
49    def errorHandler(self, error, validate=False, diff=None):
50      logging.error(error)
51      self.transport.write("> {}{}\n".format(diff if diff else error,
52                                             "" if validate else "\n"))
53
54    def infoHandler(self, info):
55      logging.info(info)
56
57    def lineReceived(self, line):
58      self.infoHandler("LINE RECEIVED: {}".format(line))
59      try:
60        items = line.split()
61        return {
62            "IAMAT": self.processIAMAT,
63            "WHATSAT": self.processWHATSAT,
64            "AT": self.processAT,
65            "help": lambda : self.help(),
66            "quit": lambda : self.help(quit=True)
67        }[items[0]](*items[1:])
68      except Exception as e:
69        if isinstance(e, utils.ValidationError):
70          self.errorHandler(e, validate=True)
71        self.commandFailed(" ".join(items))
72
73    def stopPropagation(self, client_id, client_time):
74      return client_id in self.factory.clients and self.factory.clients[
75          client_id] and float(
76              client_time) <= self.factory.clients[client_id]["time"]
77
78    def propagate(self, data, exclude=""):
79      data["response"] += " {}".format(self.factory.server_name)
80      valid_neighbors = set(
81          conf.SERVER_NEIGHBORS[self.factory.server_name]) - set(exclude)
82      for neighbor in valid_neighbors:
83        if neighbor in self.factory.connected_servers:
84          # Queue messages that failed to send
85          self.factory.connected_servers[neighbor].sendAT(data["response"])
86        else:
87          reactor.connectTCP(conf.SERVER_CONFIG[neighbor]["ip"],
88                             conf.SERVER_CONFIG[neighbor]["port"],
89                             ProxyClientFactory(self.factory, neighbor, data))
```

```
      self.infoHandler("Location update attempted from {} to {}".format(
          self.factory.server_name, neighbor))

  @utils.validate(utils.validateIAMAT)
  def processIAMAT(self, client_id, client_position, client_time):
    response = self.at_response.format(time.time() - float(client_time),
                                       client_id, client_position, client_time)
    self.transport.write("{}\n\n".format(response))
    self.infoHandler("IAMAT RESPONSE: {}".format(response))

    # Don't propagate duplicate or outdated requests
    if self.stopPropagation(client_id, client_time):
      self.infoHandler("IAMAT PROCESS: Not propagating duplicate/outdated data")
      return

    client_position = ",".join(re.findall(r"([-+]\d+\.\d+)", client_position))
    self.factory.clients[client_id] = {
        "response": response,
        "position": client_position,
        "time": float(client_time)
    }
    self.propagate(self.factory.clients[client_id])

  @utils.validate(utils.validateWHATSAT)
  def processWHATSAT(self, client_id, radius, bound):

    def processGooglePlacesQuery(response, at_message, bound):
      try:
        response_json = json.loads(response)
        response_json["results"] = response_json["results"][:bound]
      except:
        error_ = "WHATSAT PROCESS: Error processing Google Places request JSON"
        self.errorHandler("{}: {}".format(error_, response_json), diff=error_)
      else:
        client_message = "{}\n{}\n\n".format(
            at_message, json.dumps(response_json, indent=2))
        log_message = "{}\n{}".format(at_message, json.dumps(response_json))
        self.transport.write(client_message)
        self.infoHandler("WHATSAT RESPONSE: {}".format(log_message))

    def processGooglePlacesQueryError(error):
      error_ = "WHATSAT PROCESS: Error processing Google Places Request"
      self.errorHandler("{}: {}".format(error_, error.parents[0]), diff=error_)

    if client_id not in self.factory.clients or not self.factory.clients[
        client_id]:
      self.errorHandler(
          "WHATSAT REQUEST: There is no location data at server {} for client {}"
          .format(self.factory.server_name, client_id))
      return

    at_message = self.factory.clients[client_id]["response"]
    coordinates = self.factory.clients[client_id]["position"]

    query_url = conf.GOOGLE_PLACE_API_URL.format(coordinates, radius)
    self.infoHandler("Google Places query URL: {}".format(query_url))

    query_response = getPage(query_url) \
        .addCallback(processGooglePlacesQuery, at_message, int(bound)) \
```

```
        .addErrback(processGooglePlacesQueryError) \
        .addTimeout(1.0, reactor)

  @utils.validate(utils.validateAT)
  def processAT(self, server_name, time_difference, client_id, client_position,
                client_time, sender_name):
    if self.stopPropagation(client_id, client_time):
      self.infoHandler("AT PROCESS: Not propagating duplicate/outdated data")
      return

    response = self.at_response.format(time.time() - float(client_time),
                                       client_id, client_position, client_time)
    client_position = ",".join(re.findall(r"([-+]\d+\.\d+)", client_position))
    self.infoHandler("Added or updated {} at {}: {}".format(
        client_id, client_time, response))
    self.factory.clients[client_id] = {
        "response": response,
        "position": client_position,
        "time": float(client_time)
    }
    self.propagate(self.factory.clients[client_id], exclude=sender_name)


# Modeled from: https://twistedmatrix.com/documents/current/core/howto/servers.html#factories
class ProxyServerFactory(protocol.ServerFactory):

  def __init__(self, server_name, server_port):
    self.server_name = server_name
    self.server_port = server_port
    self.connections = 0
    self.clients = {}
    self.connected_servers = {}

    try:
      os.makedirs("logs")
    except OSError:
      if not os.path.isdir("logs"):
        raise

    self.log_file = "logs/server-{}.log".format(self.server_name)
    logging.basicConfig(
        filename=self.log_file,
        level=logging.DEBUG,
        filemode="a",
        format="%(asctime)s %(message)s")
    logging.info("SERVER STARTED {}:{}".format(self.server_name,
                                               self.server_port))

  def buildProtocol(self, address):
    return ProxyServerProtocol(self)

  def stopFactory(self):
    logging.info("SERVER SHUTDOWN {}:{}".format(self.server_name,
                                                self.server_port))
```

client.py

```python
#!/usr/bin/env python

# CS 131 - Twisted Project
# Client Implementation

import logging

from twisted.internet import protocol
from twisted.protocols.basic import LineReceiver


class ProxyClientProtocol(LineReceiver):

  def __init__(self, factory):
    self.factory = factory

  def connectionMade(self):
    self.factory.server.connected_servers[
        self.factory.server_name] = self.factory
    logging.info("Connection made from {} to {}".format(
        self.factory.server.server_name, self.factory.server_name))
    self.sendLine(self.factory.data["response"])

  def connectionLost(self, reason):
    # protocol.connectionLost called before factory.clientConnectionLost
    # perhaps due to client disconnecting to server rather than vice-versa
    if self.factory.server_name in self.factory.server.connected_servers:
      del self.factory.server.connected_servers[self.factory.server_name]
      logging.info("Connection lost from {} to {}".format(
          self.factory.server.server_name, self.factory.server_name))


# Modeled from: https://twistedmatrix.com/documents/current/core/howto/clients.html#clientfactory
class ProxyClientFactory(protocol.ClientFactory):

  def __init__(self, server, server_name, data):
    # NOTE: This adds a circular server-client dependency but this isn't an
    #       issue because ProxyClientProtocol disconnects before
    #       ProxyClientFactory
    self.server = server
    self.server_name = server_name
    self.data = data

  def buildProtocol(self, addr):
    self.protocol = ProxyClientProtocol(self)
    return self.protocol

  def sendAT(self, at_message):
    self.protocol.sendLine(at_message)

  def clientConnectionLost(self, connector, reason):
    if self.server_name in self.server.connected_servers:
      del self.server.connected_servers[self.server_name]
      logging.info("Connection lost from {} to {}".format(
          self.server.server_name, self.server_name))

  def clientConnectionFailed(self, connector, reason):
```

```
58      logging.info("Connection failed from {} to {}".format(
59          self.server.server_name, self.server_name))
```

utils.py

```
1   #!/usr/bin/env python
2
3   # CS 131 - Twisted Project
4   # Utils for input validation
5
6   import re
7
8
9   class ValidationError(Exception):
10    pass
11
12
13  def validate(validationHandler):
14    def real_decorator(func):
15      def wrapper(self, *args, **kwargs):
16        try:
17          validationHandler(args, **kwargs)
18        except:
19          raise
20        else:
21          func(self, *args, **kwargs)
22      return wrapper
23    return real_decorator
24
25
26  def validateIAMAT(data, check="IAMAT"):
27    if len(data) != 3:
28      raise ValidationError("{} LENGTH: {} does not have three fields".format(
29          check, data))
30    try:
31      validateGPS(data[1], check=check)
32    except:
33      # Exception formatting done in validateGPS function
34      raise
35    try:
36      time = float(data[2])
37    except:
38      raise ValidationError("{} TIME: {} is not a valid POSIX time".format(
39          check, data[2]))
40
41
42  def validateWHATSAT(data):
43    if len(data) != 3:
44      raise ValidationError(
45          "WHATSAT LENGTH: {} does not have three fields".format(data))
46    try:
47      if int(data[1]) < 0 or 50 < int(data[1]):
48        raise ValidationError(
49            "WHATSAT RADIUS: {} not in valid range (0, 50)".format(data[1]))
50      if int(data[2]) < 0 or 20 < int(data[2]):
51        raise ValidationError(
52            "WHATSAT BOUND: {} not in valid range (0, 20)".format(data[2]))
53    except:
```

```
    raise ValidationError(
        "WHATSAT ARGS: {} or {} are not valid integer parameter(s)".format(
            data[1], data[2]))


def validateAT(data):
  if len(data) != 6:
    raise ValidationError("AT LENGTH: {} does not have six fields".format(data))
  try:
    client_time = float(data[1])
  except:
    raise ValidationError("AT TIME: {} is not valid POSIX time".format(data[1]))
  try:
    validateIAMAT(data[2:-1], check="AT")
  except:
    # Exception formatting done in validateIAMAT function
    raise


def validateGPS(data, check):
  try:
    coords = re.findall(r"([-+]\d+\.\d+)", data)
    lat, lon = map(float, coords)
    if lat < -90 or 90 < lat or lon < -180 or 180 < lon:
      raise ValidationError(
          "{} GPS FORMAT: {} are not valid coordinates per ISO 6709".format(
              check, ",".join(coords)))
  except:
    raise ValidationError(
        "{} GPS FORMAT: {} is not a correctly formatted pair of coordinates".
        format(check, data))
```